Functional Notation and Lazy Evaluation in Ciao

Amadeo Casas[1] Daniel Cabeza[2] Manuel Hermenegildo[1,2]

amadeo@cs.unm.edu, herme@unm.edu,
{dcabeza, herme}@fi.upm.es

[1]Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, Albuquerque, NM, USA.
[2]School of Computer Science, T. U. Madrid (UPM), Madrid, Spain

Abstract

Certain aspects of functional programming provide syntactic convenience, such as having a designated implicit output argument, which allows function call nesting and sometimes results in more compact code. Functional programming also sometimes allows a more direct encoding of lazy evaluation, with its ability to deal with infinite data structures. We present a syntactic functional extension of Prolog covering function application, predefined evaluable functors, functional definitions, quoting, and lazy evaluation. The extension is also composable with higher-order features. We also highlight the Ciao features which help implementation and present some data on the overhead of using lazy evaluation with respect to eager evaluation.

Keywords: Declarative Languages, Logic Programming, Functional Programming, Logic-Functional Programming, Lazy Evaluation.

1 Introduction

Logic Programming offers a number of features, such as nondeterminism and partially instantiated data structures, that give it expressive power beyond that of functional programming. However, certain aspects of functional programming provide in turn syntactic convenience. This includes for example having a syntactically designated output argument, which allows the usual form of function call nesting and sometimes results in more compact code. Also, lazy evaluation, which brings the ability to deal with infinite (non-recursive) data structures [22, 1], while subsumed by logic programming features such as delay declarations, enjoys a more direct encoding in functional programming. Bringing this syntactic convenience to logic programming can result in a more compact program representation in certain cases and is therefore a desirable objective.

With this objective in mind, in this paper we present a design for an extensive functional syntactic layer for logic programs and its implementation in the Ciao system [4].[1] While the idea of adding functional features to logic programming systems is clearly not new [2, 3, 21], and there are currently a good number of systems which integrate functions into some form of logic programming, such as Oz [16] and Mercury [25] (developed simultaneously and independently from Ciao), or perform a full integration of functional and logic programming, such as Curry [15, 20], we feel that our proposal and its implementation has peculiarities which make it interesting in itself, for its application to standard Prolog systems or otherwise.

[1]The design described herein actually includes some minor changes with respect to the current version which will be available in the upcoming release of Ciao.

Our target system, Ciao, offers a complete Prolog system, supporting ISO-Prolog, with a novel modular design [9] which allows both restricting and extending the language. This design has allowed us to add functional features to it completely at the source (Prolog) level, and without modifying at all the compiler or the low-level machinery. The implementation is made by means of a number of Ciao *packages* [9], i.e., through the same extension mechanism by which other syntactic and semantic extensions are supported in Ciao, including constraints, objects, feature terms/records, persistence, several control rules, etc., giving Ciao its multi-paradigm flavor. However, our approach to the implementation of functions is actually also relatively straightforward to implement in any modern Prolog system [14]. We will highlight the Ciao features which make implementation in our view smoother and more orthogonal with the upfront intention of contributing if possible to the design of other logic programming systems, which is certainly an important objective of Ciao.

The rest of the paper is organized as follows: first, we discuss in Section 2 our general approach to integrating functional notation in a logic programming system. Section 3 presents how we implemented this approach in Ciao. Section 4 shows an example of the use of lazy evaluation, and how it is achieved by our implementation. Section 5 presents some experimental results. We come to a close in Section 6 with our conclusions.

2 Functional Notation in Ciao

2.1 Basic Concepts and Notation

Our notion of functional notation for logic programming departs in a number of ways from previous proposals. The fundamental one is that functional notation simply provides *syntactic sugar* for defining and using predicates as if they were functions, but they can still retain the power of predicates. In this model, any function definition is in fact defining a predicate, and any predicate can be used as a function. The predicate associated with a function has the same name and one more argument, meant as the place holder for the result of the function. This argument is by default added to the right, i.e., it is the last argument, but can be defined otherwise by using the appropriate declaration. The syntax extensions provided for functional notation are the following:

Function applications Any term preceded by the `~` operator is a function application, as can be seen in the goal `write(~arg(1,T))`, which is equivalent to the sequence `arg(1,T,A), write(A)`. To use a predicate argument other than the last as the return argument, a declaration like "`:- fun_return functor(~,_,_).`" can be used, so that `~functor(f,2)` is evaluated to `f(_,_)` (where `functor/3` is the standard ISO-Prolog builtin). This definition of the return argument can also be done on the fly in each invocation in the following way: `~functor(~,f,2)`. Functors can be declared as *evaluable* (i.e., being in calls in functional syntax) by using the declaration `function/1`. This allows avoiding the need to use the `~` operator. Thus, "`:- function arg/2.`" allows writing `write(arg(1,T))` instead of `write(~arg(1,T))` as above. This declaration can be combined with the previous one: "`:- function functor(~,_,_).`". Note that all these declarations, as is customary in Ciao, are local to the module where they are included. Finally, the `~` operator does not need to be used within a functional definition (see below) for the functor being defined.

Predefined evaluable functors In addition to functors declared with the declaration `function/1`, several functors are evaluable by default, those being:

- All the functors understood by `is/2`. Thus, it is possible to use `write(A+B)` to denote `T is A+B, write(T)`. This feature can be disabled by a declaration `:- function arith(false)` (and reverted by using `true` instead of `false`).
- The functors used for disjunctive and conditional expressions, `(|)/2` and `(?)/2`. A disjunctive expression has the form `(V1|V2)`, and its value when first evaluated is `V1`, and on backtracking `V2`. A conditional expression has the form `(Cond ? V1)`, or more commonly `(Cond ? V1 | V2)`, and its value, if the execution of `Cond` as a goal succeeds, is `V1`, otherwise in the first form it causes backtracking, and on the second form its value is `V2`. Due to operator precedences, a nested expression `(Cond1 ? V1 | Cond2 ? V2 | V3)` is evaluated as `(Cond1 ? V1 | (Cond2 ? V2 | V3))`.

Functional definitions A functional definition is composed of one of more functional clauses. A functional clause is written using the binary operator `:=`, as in

```
opposite(red) := green.
```

Functional clauses can also have a body, which is executed before the result value is computed. It can serve as a guard for the clause or to provide the equivalent of where-clauses in functional languages:

```
fact(0) := 1.
fact(N) := N * fact(--N) :- N > 0.
```

In a functional clause, the defined function does not need to be preceded by `~`. Note that guards can often be defined more compactly using conditional expressions:

```
fact(N) := N = 0 ? 1
         | N > 0 ? N * fact(--N).
```

The translation of functional clauses defining recursive predicates maintains the tail recursion of the equivalent predicate, thus allowing the usual compiler optimizations.

Quoting functors In clause heads (independently of whether they are defined as predicates or functions) functors can be prevented from being evaluated by using the `(^)/1` prefix operator (read as "quote"), as in

```
pair(A,B) := ^(A-B).
```

Note that this just prevents the evaluation of the principal functor of the enclosed term, not the possible occurrences of other evaluable functors inside.

Scoping When using function applications inside the goal arguments of meta-predicates, there is an ambiguity as they could be evaluated either in the scope of the outer execution or the in the scope of the inner execution. The chosen behavior is by default to evaluate function applications in the scope of the outer execution, and if they should be evaluated in the inner scope, the goal containing the function application needs to be escaped with the `(^^)/1` prefix operator, as in `findall(X, (d(Y), ^^(X = ~f(Y)+1)), L)` (which could also be written as `findall(X, ^^ (d(Y), X = ~f(Y)+1), L)`).

Laziness Lazy evaluation is a program evaluation technique used particularly in functional languages. When using lazy evaluation, an expression is not evaluated as soon as it is assigned, but rather when the evaluator is forced to produce the value of the expression. The `when`, `freeze`, or `block` control primitives present in many modern logic programming systems are

more powerful than lazy evaluation. However, they lack the simplicity of use and cleaner semantics of functional lazy evaluation. In our design, a function (or predicate) can be declared as lazy via the declarations: "`:- lazy function function_name/N.`". (or, equivalently in predicate version, "`:- lazy pred_name/M.`", where $M = N + 1$). In order to achieve the intended behavior, the execution of each function declared as lazy is suspended until the return value of the function is needed.

Definition of real functions In the previous scheme, functions are (at least by default) not forced to provide a single solution for their result, and, furthermore, they can be partial, producing a failure when no solution can be found. A predicate defined as a function can be declared to behave as a real function using the declaration "`:- funct name/N.`". Such predicates are then converted automatically to real functions by adding pruning operators and a number of Ciao assertions [23] which pose (and check) additional restrictions such as determinacy, modedness, etc., so that the semantics will be the same as in traditional functional programming.

2.2 Examples

We now illustrate the use of the functionality introduced above through examples. The following example defines a simple unary function `der(X)` which returns the derivative of a polynomial arithmetic expression:

```
der(x)          := 1.
der(C)          := 0                        :- number(C).
der(^(A + B))   := ^(der(A) + der(B)).
der(^(C * A))   := ^(C * der(A))            :- number(C).
der(^(x ** N)) := ^(N * ^(x ** (N - 1))) :- integer(N), N > 0.
```

Note that the code is a bit obfuscated because it uses frequently evaluable functors which need to be quoted. This quoting can be prevented by including the directive mentioned above to make functors understood by `is/2` not evaluable. Thus, the arithmetic evaluation in the last clause needs to be explicitly requested. The new program follows:

```
:- function(arith(false)).
der(x)      := 1.
der(C)      := 0                  :- number(C).
der(A + B)  := der(A) + der(B).
der(C * A)  := C * der(A)         :- number(C).
der(x ** N) := N * x ** ~(N - 1) :- integer(N), N > 0.
```

Both of the previous code fragments translate to the following code:

```
der(x, 1).
der(C, 0) :-
          number(C).
der(A + B, X + Y) :-
          der(A, X),
          der(B, Y).
der(C * A, C * X) :-
          number(C),
```

```
          der(A, X).
der(x ** N, N * x ** N1) :-
          integer(N),
          N > 0,
          N1 is N - 1.
```

Note that with our solution the programmer may use `der/2` as a function or as a predicate indistinctly.

As a simple example of the use of lazy evaluation consider the following definition of a function which returns the (potentially) infinite list of integers starting with a given one:

```
:- lazy function nums_from/1.
nums_from(X) := [X | nums_from(X+1)].
```

One of the nice features of functional notation is that it allows writing regular types (used in Ciao assertions [23] and checked by the preprocessor) in a very compact way:

```
color := red | blue | green.
list := [] | [_ | list].
list_of(T) := [] | [~T | list_of(T)].
```

which are equivalent to (note the use of higher-order in the third example, to which we will refer again later):

```
color(red). color(blue). color(green).
list([]). list([_|T]) :- list(T).
list_of(_, []). list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

3 Implementation Details

As mentioned previously, certain Ciao Prolog features have simplified its extension to handle functional notation. In the following we sketch the features of Ciao we used and then how we applied them for our needs.

3.1 Code Translations in Ciao

Traditionally, Prolog systems have included the possibility of changing the syntax of the source code by the use of the `op/3` builtin/directive. Furthermore, in many Prolog systems it is also possible to define *expansions* of the source code (essentially, a very rich form of "macros") by allowing the user to define (or extend) a predicate typically called `term_expansion/2` [24, 12]. This is usually how, e.g., definite clause grammars (DCG's) are typically implemented.

However, these features, in their original form, pose many problems for modular compilation or even for creating sensible standalone executables. First, the definitions of the operators and, specially, expansions are global, affecting a number of files. Furthermore, which files are affected cannot be determined statically, because these features are implemented as a side-effect, rather than a declaration, and they are meant to be active after they are read by the code processor (top-level, compiler, etc.) and remain active from then on. As a result, it is impossible by looking at a source code file to know if it will be affected by expansions or definitions of operators, which may completely change what the compiler really sees.

In order to solve these problems, these features were redesigned in Ciao, so that it is still possible to define source translations and operators, but such translations are local to the module or user

file defining them. Also, these features are implemented in a way that has a well-defined behavior in the context of a stand-alone compiler (and this has been applied in the Ciao compiler, `ciaoc` [8]). In particular, the directive `load_compilation_module/1` allows separating code that will be used at compilation time (e.g., the code used for translations) from code which will be used at run-time. It loads the module defined by its argument *into the compiler*.

In addition, in order to make the task of writing expansions easier, the effects usually achieved through `term_expansion/2` can be obtained in Ciao by means of four different, more specialized directives, which, again, *affect only the current module*.

The solutions we provided to implement functions in Ciao are based on the possibility of defining these source translations. In particular, we have used the directives `add_sentence_trans/1` (to define *sentence translations*) and `add_goal_trans/1` (to define *goal translations*). A sentence translation is a predicate which will be called by the compiler to possibly convert each term read by the compiler to a new term, which will be used by the compiler in place of the original term. A goal translation is a predicate which will be called by the compiler to possibly convert each goal present in the clauses of the current text to another goal to replace the original one. Note the proposed model can be implemented in other Prolog systems using similarly using `term_expansion/2` and operator declarations, but that having operators and syntactic transformation predicates local to modules makes it scalable and amenable to combination with other packages and syntactic extensions.

3.2 Ciao Packages

Packages in Ciao are libraries which define extensions to the language, and have a well defined and repetitive structure. These libraries typically consist of a main source file which defines only some declarations (operator declarations, declarations loading other modules into the compiler or the module using the extension, etc.). This file is meant to be *included* as part of the file using the library, since, because of their local effect, such directives must be part of the code of the module which uses the library. Any auxiliary code needed at compile-time (e.g., translations) is included in a separate module which is to be loaded into the compiler via a `load_compilation_module/1` directive placed in the main file. Also, any auxiliary code to be used at run-time is placed in another module, and the corresponding `use_module` declaration is also placed in the include file.

In our implementation of functional notation in Ciao we have provided two packages: one for the bare function features without lazy evaluation, and an additional one to provide the lazy evaluation features. The reason for this is that in many cases the lazy evaluation features are not needed and thus the translation procedure is simplified.

3.3 The Ciao Implementation of Functional Extensions

To translate the functional definitions, we have used as mentioned above the `add_sentence_trans/1` directive to provide a translation procedure which transforms each functional clause to a predicate clause, adding to the function head the output argument, in order to convert it to the predicate head. This translation procedure also deals with functional applications in heads, as well as with `function` directives.

On the other hand, we have used the `add_goal_trans/1` directive to provide a translation procedure for dealing with function applications in bodies. The rationale to using a goal translation is that each function application inside a goal will be replaced by a variable, and the goal will be preceded by a call to the predicate which implements the function in order to provide a value for that variable.

An additional sentence translation is provided to handle the `lazy` directives. The translation of a lazy function into a predicate is done in two steps. First, the function is converted into a predicate using the procedure explained above. Then, the resulting predicate is transformed to suspend its execution until the value of the output variable is needed. This suspension is achieved by the use of the `freeze/1` control primitive that many modern logic programming systems implement quite efficiently [11] (`block` or `when` declarations can obviously also be used, but we explain the transformation in terms of `freeze` because it is more widespread). This translation will rename the original predicate to an internal name and add a *bridge predicate* with the original name which invokes the internal predicate through a call to `freeze/1`. This will delay the execution of the internal predicate until its result is required, which will be detected as a binding (i.e., demand) of its output variable. The following section will provide a detailed example of the translation of a lazy function. The implementation with `block` is even simpler since no bridge predicate is needed.

We show below, for reference, the main files for the Ciao library packages `functions`:

```
% functions.pl
:- include(library('functions/ops')).
:- load_compilation_module(library('functions/functionstr')).
:- add_sentence_trans(defunc/3).
:- add_goal_trans(defunc_goal/3).
```

and `lazy` (which will usually be used in conjunction with the first one):

```
% lazy.pl
:- include(library('lazy/ops')).
:- use_module(library(freeze)).
:- load_compilation_module(library('lazy/lazytr')).
:- add_sentence_trans(lazy_sentence_translation/3).
```

The Ciao system source provides the actual detailed coding along the lines described above.

4 Lazy Functions: an Example

In this section we show an example of the use of lazy evaluation, and how a lazy function is translated by our Ciao package.

Figure 1 shows in the first row the definition of a lazy function which returns the infinite list of Fibonacci numbers, in the second row its translation into a lazy predicate[2] (by the `functions` package) and in the third row the expansion of that predicate to emulate lazy evaluation (where `fiblist_$$lazy$$` stands for a fresh predicate name).

In the `fiblist` function defined, any element in the resulting *infinite* list of Fibonacci numbers can be referenced, as for example, `nth(X, ~fiblist, Value)`. The other functions used in the definition are `tail/2`, which is defined as lazy and returns the tail of a list; `zipWith/3`, which is also defined as lazy and returns a list whose elements are computed by a function having as arguments the successive elements in the lists provided as second and third argument;[3] and `add/2`, defined as `add(X,Y) := X + Y`.

Note that the `zipWith/3` function (respectively the `zipWith/4` predicate) is in fact a *higher-order* function (resp. predicate). Ciao provides in its standard library a package to support higher-order [7, 5, 10], which together with the `functions` and `lazy` packages (and, optionally, the type inference and checking provided by the Ciao preprocessor [18]) basically provide all the functionality present in modern functional languages.

[2] The `:- lazy function fiblist/0.` declaration is converted into a `:- lazy fiblist/1.` declaration.

[3] It has the same semantics as the `zipWith` function in Haskell.

```
:- lazy function fiblist/0.
fiblist := [0, 1 | ~zipWith(add, FibL, ~tail(FibL))]
        :- FibL = fiblist.
```

```
:- lazy fiblist/1.
fiblist([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(add, FibL, T, Rest).
```

```
fiblist(X) :-
          freeze(X, 'fiblist_$$lazy$$'(X)).

'fiblist_$$lazy$$'([0, 1 | Rest]) :-
          fiblist(FibL),
          tail(FibL, T),
          zipWith(add, FibL, T, Rest).
```

Figure 1: Code translation for a fibonacci function, to be evaluated lazily.

5 Some Performance Measurements

Since the functional extensions proposed simply provide a syntactic bridge between functions and predicates, there are only a limited number of performance issues worth discussing. For the case of *real* functions, it is well known that performance gains can be obtained from the knowledge of a certain function or predicate being moded (e.g., for a function all arguments being ground on input and the "designated output" argument being ground on output), determinate, non-failing, etc. [26, 17, 19]. In Ciao this information can in general (i.e., for any predicate or function) be inferred by the Ciao preprocessor or declared with Ciao assertions [18, 23]. As mentioned before, for declared "real" (`func`) functions, the corresponding information is added automatically. Some results on current Ciao performance when this information is available are presented in [19].

In the case of lazy evaluation of functions, the main goal of the technique presented herein is not really any increase in performance, but achieving new functionality and convenience through the use of code translations and delay declarations. However, while there have also been some studies of the overhead introduced by delay declarations, it is interesting to see how this overhead affects our implementation of lazy evaluation by observing its performance. Consider the `nat/2` function in Figure 2, a simple function which returns a list with the first N numbers from an (infinite) list of natural numbers.

```
:- function nat/1.
nat(N) := ~take(N, nums_from(0)).

:- lazy function nums_from/1.
nums_from(X) := [X | nums_from(X+1)].
```

```
:- function nat/1.
:- function nats/2.
nat(X) := nats(0, X).
nats(X, Max) := X > Max ? []
              | [X | nats(X+1, Max)].
```

Figure 2: Lazy and eager versions of function `nat(X)`.

Function `take/2` in turn returns the list of the first N elements in the input list. This `nat(N)` function cannot be directly executed eagerly due to the infinite list provided by the `nums_from(X)` function, so that, in order to compare time and memory results between lazy and eager evaluation, an equivalent version of that function is provided.

Table 1 reflects the time and memory overhead of the lazy evaluation version of `nat(X)` and that of the equivalent version executed eagerly. As a further example, Table 2 shows the results for a quicksort function executed lazily in comparison to the eager version of this algorithm. All the results were obtained by averaging ten runs on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 2.0, and with the translation of Figure 1.

	Lazy Evaluation		**Eager Evaluation**	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.030	1503.2	0.002	491.2
100 elements	0.276	10863.2	0.016	1211.2
1000 elements	3.584	104463.0	0.149	8411.2
2000 elements	6.105	208463.2	0.297	16411.2
5000 elements	17.836	520463.0	0.749	40411.2
10000 elements	33.698	1040463.0	1.277	80411.2

Table 1: Performance for `nat/2` (time in ms. and heap sizes in bytes).

	Lazy Evaluation		**Eager Evaluation**	
List	**Time**	**Heap**	**Time**	**Heap**
10 elements	0.091	3680.0	0.032	1640.0
100 elements	0.946	37420.0	0.322	17090.0
1000 elements	13.303	459420.0	5.032	253330.0
5000 elements	58.369	2525990.0	31.291	1600530.0
15000 elements	229.756	8273340.0	107.193	5436780.0
20000 elements	311.833	11344800.0	146.160	7395100.0

Table 2: Performance for `qsort/2` (time in ms. and heap sizes in bytes).

We can observe that there is certainly an impact on the execution time when functions are evaluated lazily, but even with this version the results are quite acceptable if we take into account that the execution of the predicate does really suspend. Related to memory consumption we show heap sizes, without garbage collection (in order to observe the raw memory consumption rate). Lazy evaluation implies as expected some memory overhead due to the need to copy (freeze) program goals into the heap. Also, while comparing with standard lazy functional programming implementations is beyond the scope of this paper, some simple tests done for sanity check purposes (with HUGS) show that the results are comparable, our implementation being for example slower on `nat` but faster on `qsort`, presumably due to the different optimizations being performed by the compilers.

An example when lazy evaluation can be a better option than eager evaluation in terms of performance and not only convenience is found in a concurrent or distributed system environment (such as, e.g., [13]), and in the case of Ciao also within the active modules framework [4, 6]. The example in Figure 3 uses a function, defined in an active module, which returns a big amount of data. Function `test/0` in module `module1` needs to execute function `squares/1`, in (active, i.e., remote)

module `module2`, which will return a very long list (which could be infinite for our purposes). If `squares/1` were executed eagerly then the entire list would be returned, to immediately execute the `takeWhile/2` function with the entire list. `takeWhile/2` returns the first elements of a (possibly infinite) list while the specified condition is true. But creating the entire initial list is very wasteful in terms of time and memory requirements. In order to solve this problem, the `squares/1` function could be moved to module `module1` and merged with `takeWhile/2` (or, also, they could exchange a size parameter). But rearranging the program is not always possible and may perhaps complicate other aspects of the overall program design.

But if the `squares/1` function is evaluated lazily, it is possible to keep the definitions unchanged and in different modules, so that there will be a smaller time and memory penalty for generating and storing the intermediate result. As more values are needed by the `takeWhile/2` function, more values in the list returned by `squares/1` are built (in this example, only while the new generated value is less than 10000), considerably reducing the time and memory consumption that the eager evaluation would take.

```
:- module(module1, [test/1], [functions, lazy, hiord,actmods]).
:- use_module(library('actmods/webbased_locate')).

:- use_active_module(module2, [squares/2]).

:- function takeWhile/2.
takeWhile(P, [H|T]) := P(H) ? [H | takeWhile(P, T)]
                            | [].
:- function test/0.
test := takeWhile(condition, squares).
condition(X) :- X < 10000.
```

```
:- module(module2, [squares/1], [functions, lazy, hiord]).

:- lazy function squares/0.
squares := map_lazy(take(1000000, nums_from(0)), square).

:- lazy function map_lazy/2.
map_lazy([], _)     := [].
map_lazy([X|Xs], P) := [~P(X) | map_lazy(Xs, P)].

:- function take/2.
take(0, _)     := [].
take(X, [H|T]) := [H | take(X-1, T)] :- X > 0.

:- lazy function nums_from/1.
nums_from(X) := [X | nums_from(X+1)].

:- function square/1.
square(X) := X * X.
```

Figure 3: A distributed (active module) application using lazy evaluation.

6 Conclusions

We have presented a functional extension of Prolog, which includes the possibility of evaluating functions lazily. The proposed approach has been implemented in Ciao and is used now throughout the libraries and other system code (including the preprocessor and documenter) as well as in a number of applications written by the users of the system.

The performance of the package has been tested with several examples. As expected, evaluating functions lazily implies some overhead (and also additional memory consumption) with respect to eager evaluation. This justifies letting the user choose via annotations for which functions or predicates lazy evaluation is desired. In any case, the main advantage of lazy evaluation is to make it easy to work with infinite (non-periodic) data structures in the manner that is familiar to functional programmers.

References

[1] S. Antoy. Lazy evaluation in logic. In *Symp. on Progr. Language Impl. and Logic Progr (PLILP'91)*, pages 371–382. Springer Verlag, 1991. LNCS 528.

[2] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *International Symposium on Logic Programming*, pages 160–168, Silver Spring, MD, February 1984. IEEE Computer Society.

[3] G. L. Bellia, M. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[4] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. P. (Eds.). The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM), June 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[5] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[6] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.clip.dia.fi.upm.es/`.

[7] D. Cabeza and M. Hermenegildo. Higher-order Logic Programming in Ciao. Technical Report CLIP7/99.0, Facultad de Informática, UPM, September 1999.

[8] D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147–164. N.M. State U., December 1999.

[9] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[10] D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

[11] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.

[12] M. Carlsson and J. Widen. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, April 1994.

[13] M. Carro and M. Hermenegildo. A simple approach to distributed objects in prolog. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[14] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[15] M. Hanus et al. Curry: An Integrated Functional Logic Language. `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.

[16] S. Haridi and N. Franzén. *The Oz Tutorial*. DFKI, February 2000. Available from `http://www.mozart-oz.org`.

[17] F. Henderson et al. *The Mercury Language Reference Manual*. URL: `http://www.cs.mu.oz.au/research/mercury/information/doc/ reference_manual_toc.html`.

[18] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.

[19] J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3507 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.

[20] J. Moreno Navarro and M. Rodríguez-Artalejo. BABEL: A functional and logic programming language based on constructor discipline and narrowing. In *Conf. on Algebraic and Logic Programming (ALP)*, LNCS 343, pages 223–232, 1989.

[21] L. Naish. Adding equations to NU-Prolog. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, number 528 in Lecture Notes in Computer Science, pages 15–26, Passau, Germany, August 1991. Springer-Verlag.

[22] S. Narain. Lazy evaluation in logic programming. In *Proc. 1990 Int. Conference on Computer Languages*, pages 218–227, 1990.

[23] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[24] Quintus Computer Systems Inc., Mountain View CA 94041. *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[25] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[26] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.